A SIMPLE, FAST, AND EFFECTIVE LL(1) ERROR REPAIR ALGORITHM

by

Charles N. Fischer and Jon Mauney

Computer Sciences Technical Report #901

December 1989

A Simple, Fast, and Effective LL(1) Error Repair Algorithm

Charles N. Fischer
Computer Sciences Department
University of Wisconsin–Madison

Jon Mauney
Computer Science Department
North Carolina State University

November 16, 1989

Abstract

Validation and locally least-cost repair are two simple and effective techniques for dealing with syntax errors. We show how the two can be combined into an efficient and effective error-handler for use with LL(1) parsers. Repairs are computed using an extension of the FMQ algorithm. Tables are created as necessary, rather than precomputed, and possible repairs are kept in a priority queue. Empirical results show that the repairs chosen with this strategy are of very high quality and that speed is quite acceptable.

1 Introduction

The problem of handling syntax errors during context-free parsing has been widely studied and many approaches have been published. None, however, has achieved widespread acceptance. The FMQ algorithm [3], a locally least-cost repair method for use with LL(1) parsers, has several advantages. The algorithm is very simple and efficient, it is based on a high-level model of least-cost repair, the error-handler can be automatically generated from a context-free grammar, and the costs can easily be tuned to give good quality repairs. FMQ suffers from two drawbacks. First, it depends on a fairly large precomputed table (39k bytes for Pascal, 85k bytes for Ada). Second, there are many situations in which the locally least-cost model does not produce acceptable repairs; most of these could be handled by adding a simple *validation* phase to the repair algorithm, but FMQ is not directly amenable to validation.

We show how these two problems can be resolved by extending and generalizing the table used by FMQ and computing on the fly only those elements of the table that are needed. The result is an error-repair scheme that is simple in concept, automatically generated, efficient, and which produces extremely good repairs.

1.1 Validation

Validation is a popular adjunct to error-repair strategies. In validation, a repair is selected and instantiated, and then the parser is tentatively restarted. If the parser is able to consume a sufficient portion of the program following the error, then the repair is accepted. If however another error is detected during validation, the the repair is rejected and another candidate is tried.

Validation can be very useful in choosing from a set of possible repairs. For example, suppose a parser reads the following symbols in a statement:

i := j k

An error will be announced when the k is read, and possible repairs include inserting any of the following symbols between j and k:

+ [(;

A strictly local repair algorithm must chose among these, and other, possibilities solely on the basis of the information presented so far. A cost-based repair algorithm can be tuned to choose the repair that is most likely to correct the error, but poor repairs must result a certain percentage of the time. A validating repair strategy, on the other hand, can take advantage of information gleaned from the program text following the point of error. Continuing the example, suppose the entire line contains

i := j k,m];

A repair that inserts "+", "(", or ";" between j and k will fail to validate and be rejected, but the repair that inserts "[" will be accepted.

Validation is not a panacea, and in fact brings up some difficult problems. If there are two distinct errors near each other in a program then the second error may invalidate all attempted repairs to the first; the validater reports only the detection of an error, not the ultimate cause of the error. Also, the question arises of how far into the program to validate. A short validation may fail to gather sufficient information. In the example above, after validating over the three symbols "k , m" both "(" and "[" are still viable repairs; it is only when the fourth symbol, "]", is seen that the best repair is known. On the other hand, a long validation would be time-consuming and would provide no additional benefit in the majority of the cases. While we have investigated theoretical solutions to these problems [7, 6], a simpler, more practical approach is desirable.

A combination of least-cost repair with validation is attractive. The repair costs can be used to tune the order in which candidate repairs are tried, putting the most plausible repairs first. Speed will be improved because the most common errors will be repaired with very few validation attempts. In the case that validation does not live up to its promise, either because of multiple errors or because the validation region is too small, costs can be used to choose the most plausible repair anyway.

1.2 Locally Least-Cost Repair

The least-cost repair algorithm of [3], known as the FMQ algorithm, is easily extended to include deletions as well as insertions[1, pp 700–702]. The algorithm is efficient, but unfortunately it is ruthless about eliminating from consideration any repairs that have no chance of being *the* least-cost repair. In a validation algorithm, we will very likely need to examine the third-, fourth-, and fifth-best repairs, so we must extend the algorithm to provide them.

```
{ Let X_n ··· X_1 be the LL(1) parse stack
a is the error symbol
insert is the string to be inserted as a repair
prefix is a least-cost prefix derivable from the stack processed so far }
insert := ?;
prefix := λ;
for i := n downto 1 do
    if Cost(prefix) ≥ Cost(insert) then {no cheaper repair is possible}
        return(insert);
    if Cost(prefix E(X_i,a)) < Cost(insert) then { a cheaper repair has been found}
        insert := prefix E(X_i,a);
    prefix := prefix S(X_i);
return(insert);
```

Figure 1: Original FMQ repair algorithm

The heart of the repair algorithm is the original insert-only algorithm[3], and indeed the heart of that algorithm is a single table. It is on this table that we will concentrate.

The FMQ algorithm relies on the fact that the LL(1) parsing stack contains a straightforward prediction of the form of the remainder of the input. The parser announces error when the next input symbol does not immediately fit with that prediction. The FMQ algorithm works by finding a position within the parse stack at which the error-symbol *can* fit, and then inserting symbols that will bring the parser to that point.

FMQ depends on two tables, S and E, defined as follows:

- $S(A) \equiv x \in V_t^*$ such that $A \Rightarrow^* x$ and Cost(x) is minimized
- $E(A, a) \equiv x \in V_t^*$ such that $A \Rightarrow^* xay$ and Cost(x) is minimized.

We will use λ to denote the empty string, and ? will represent a special string that indicates no insertion is possible. The original FMQ algorithm is shown in figure 1.

2 Extending the FMQ algorithm

The S table is merely used to satisfy symbols on the parse stack that are not directly involved with the error-symbol. The E table, on the other hand, describes the ways the error symbol can be matched to the stack. We extend the E table to include other than least-cost matches.

- $E(A, a, k) \equiv x \in V_t^*$ such that $A \Rightarrow^* xay$ and Cost(x) is the k-th lowest

We will call k the *level* of the E-table entry. Level 0 of the extended E-table is equivalent to the original E-table.

In order to keep track of a variety of repair candidates, we maintain a priority queue. The queue is initialized during a top-to-bottom scan of the stack as in the FMQ algorithm. The least-cost repair is then removed from the queue and validated. If validation fails then the next-level E-table entry is computed

```
{ Let X_n ··· X_1 be the LL(1) parse stack
a is the error symbol}
for i := n downto 1 do   { initialize }
    if E(X_i,a,0) ≠ ? then
        add candidate (p,0) to the queue
          with cost Cost(S(X_n ... X_{i+1})) + Cost(E(X_i, a, 0))
repeat { search for repair }
    remove next candidate (i, k) from queue
    insert S(X_top ... X_{i+1})E(X_i, a, k) and validate
    if validation fails then
        add new candidate (i, k + 1) to queue
          with cost Cost(S(X_n ... X_{i+1})) + Cost(E(X_i, a, k + 1))
until success;
```

Figure 2: Extended FMQ algorithm with validation

and the resulting repair candidate added to the queue. The process repeats until a repair is successfully validated. A candidate repair can be identified by a pair (p, k) where p is the stack position and k is the level of E table entry. The repair algorithm with validation is shown in figure 2.

As with the FMQ algorithm, it is easy to add deletions to the extended algorithm. A candidate is now a triple, (p, k, d), where d is the number of symbols deleted. The cost of a candidate must of course include the delete costs of the (potentially) deleted symbols, and the error-symbol used in the E-table lookup must be the first remaining symbol after d symbols are deleted. Other than that, candidates are entered into the queue, removed, and validated as described above.

3 Incremental Computation of the E Table

In the FMQ algorithm, it is expected that the E table is precomputed. It is not feasible to precompute the extended E table, since for a given pair (A, a) there will likely be an infinite number of possible prefixes. Even if the number of prefixes is limited, the table will be large and the majority of the entries will not be used.

We will compute extended E-table entries on demand, caching values already computed to avoid unnecessary recomputation. We will assume the following objects have been precomputed and are available: P, the set of productions, Cost(a), the cost of inserting the terminal a, and S(A), the least-cost terminal string derivable from A.

A fundamental notion will be that of a *rule,* defined as the triple (p, i, k). p is a production, i is a position in p's right hand side, and k is a level. A rule can also be represented in the following (more readable) form: $A \rightarrow \alpha B^k \beta$. A rule encodes a way of computing an E-table value. In particular, the rule $A \rightarrow \alpha B^k \beta$ states that E(A,a,i) may be equal to αE(B,a,k). In fact, E-table entries are represented as rules. When a string of symbols is needed for a repair, it is easily derived from the rules and the S table. For purposes of cost computation, Cost($A \rightarrow \alpha B^k \beta$) = Cost($\alpha$) + Cost(E(B,a,k)).

```
function ComputeE0(A : V_n; a : V_t): Rule;
   (* Compute Reachable = {B ∈ V_n | A ⇒* ··· B ···} *)
for each B in Reachable do
   if E(B, a, 0) is not computed then
      Cost(E(B, a, 0)) = ∞;
      for each production p = B → X_1 ··· X_m do
         for i := 1 to m do
            if X_i in (V_n ∪ {a}) then
               Create a rule r = B → αX_i^0 β;
               Link r into 2 lists: LHS(B) and RHS(X_i);
for each X in (V_n ∪ {a}) do
   if E(X, a, 0) is computed then
      Push X onto a stack S;
while S ≠ empty do
   Pop X from S;
   for each rule r = D → αX^0 β in RHS(X) do
      if Cost(r) < Cost(E(D, a, 0)) then
         E(D, a, 0) := r;
         Push D onto S;
return E(A, a, 0);
end;
```

Figure 3: Computation of E(A,a,0)

We begin by considering how level 0 entries are computed. The algorithm ComputeE0, shown in figure 3, does this. First, a set *Reachable* of all non-terminals reachable from A (including A itself) is computed. Any of these non-terminals may be involved in a derivation from A of the error symbol, a. For each non-terminal, B, in Reachable whose E-table value is unknown, we prepare a set of rules by examining productions with B as the left-hand side. Rules are linked into two lists: LHS(B) and RHS(X). LHS(B) is simply all rules with B as the left-hand side. This list contains all the rules that might be used to compute E(B,a,0). RHS(X) contains all the rules that contain X as the selected right-hand side symbol. If the value of E(X,a,0) becomes known, it can be propagated through this list of rules.

Finally, we stack a and those reachable non-terminals whose E-table value we know. Using the RHS list, we update other E-table values. That is, if E(X,a,0) is known, and we have the rule $D \rightarrow \alpha X^0 \beta$, then αE(X,a,0) is a candidate for E(D,a,0) (if its cost is less than that of values suggested by other rules). We continue propagating new E-table values, until no new entries are found.

We next consider how to compute E(A,a,i) given that E(A,a,i-1) is already computed. We examine all the rules on LHS(A) that may define E(A,a,i), looking for that rule that yields the cheapest value. The rule that was used to compute E(A,a,i-1) must be "incremented;" that is, the level value in the rule should be increased by one. This guarantees that the value of E(A,a,i-1) is not incorrectly reused as E(A,a,i). If we increment a rule to (say) $A \rightarrow \alpha B^{j+1} \beta$, we check if E(B,a,j+1) is already computed. If it is, this rule can be examined immediately to see if it contributes a cheaper E-table value. If E(B,a,j+1) is marked as "being computed," we disregard this rule since an E-table entry cannot be used as a component in its own

```
Function ComputeE(A : V_n; a : V_t; l:level): Rule;
if E(A, a, l − 1) = ? then
return ?;
N := A; lev := l;
repeat
    Cost(E(N, a, lev)) := ∞;
    BeingComputed(N, lev) := true;
    NextLev := −1;
    for each rule r = N → αB^j β in LHS(N) do
        if r = E(N, a, lev − 1) then {Increment rule}
            if not BeingComputed(B, j) then
                Replace r with N → αB^{j+1} β;
                if E(B, a, j + 1) is computed then
                    if Cost(E(N, a, lev)) > Cost(r) then
                        E(N, a, lev) := r;
                else if not BeingComputed(B, j + 1) then
                    NextN := B;
                    NextLev := j + 1;
                    Push(r, lev) onto stack S;
        else if Cost(E(N, a, lev)) > Cost(r) then
            E(N, a, lev) := r;
    N := NextN;
    lev := NextLev;
until lev = −1;
while S ≠ empty do
    Pop (r = D → αB^j β, lev) from S;
    if Cost(E(D, a, lev)) > Cost(r) then
        E(D, a, lev) := r;
end; {ComputeE}
```

Figure 4: Computation of E-table

computation. If E(B,a,j+1) is not yet computed, we push this rule onto a stack and mark E(B,a,j+1) as a value to be computed as soon as all the rules on LHS(A) are examined. After these rules are examined, LHS(B) is examined to compute the value of E(B,a,j+1). This process continues until all needed E-table values are computed. Finally, stacked rules are examined to see if newly computed values can be used to obtain cheaper E-table values. The complete algorithm is shown in figure 4.

3.1 The Problem of Redundant Suffixes

In our repair scheme, if a candidate repair fails to validate, we compute the next level E-table entry and queue it for future consideration. It can happen that two E-table entries, of different levels, will lead to exactly the same parse-stack configuration when they are parsed. This means that if a repair based the lower-level entry (which is considered first) fails to validate, so *must* a repair based on the higher level entry. As an example, consider

IdList → Id Tail

Tail → , Id Tail | λ

E(Tail,Id,0) = "," and if ", Id" is parsed when Tail is the stack top, Tail will reappear as the stack top. Now E(Tail,Id,1) = ", Id ," and if ", Id , Id" is parsed with Tail as the stack top, again Tail reappears.

Define the *Suffix* of an E-table entry, denoted as Suf(E(A,a,i)), as the symbols that would replace A if E(A,a,i)a were parsed with A as the stack top. Suf(E(A,a,i)) is easy to obtain from the rules used to define E-table entries. In particular, if E(A,a,i) = $A \rightarrow \alpha B^j \beta$, then Suf(E(A,$a$,i)) = Suf(E(B,$a$,j))$\beta$. We will say that E(A,$a$,j) has a *redundant suffix* if Suf(E(A,a,j)) = Suf(E(A,a,i), where i < j. Experiments have shown that in practice a very significant number of E-table entries have redundant suffixes (from 47% to 74% for Pascal, from 32% to 60% for Ada). We therefore will suppress the computation and use of E-table entries that have redundant suffixes; only entries that have distinct suffixes will lead to different parse configurations.

When a new entry, E(A,a,j) is computed, we could simply compare its suffix with all entries E(A,a,i) for which i < j. In practice this is too slow, so we use the length of a suffix and a hash value based on the suffix to quickly identify suffixes that can't be equal. With these changes, we can define ComputeNonRedundantE, as shown in figure 5.

4 Implementation Results

The incremental E-table computation and validating repair algorithm were coded in Pascal and added to an LL(1) parser[1].

The parser and repair algorithm were applied to the suite of Pascal syntax errors collected by Ripley and Druseikis[8]. The resulting repairs were judged as *excellent* if a human would make the same repair, *good* if the repair is not excellent but still plausible and does not cause additional, spurious, errors to be detected, and *poor* if it is implausible or causes spurious error messages later in the program. We used a grammar with tuned repair costs as used in previous experiments [2]. All repairs were validated for five tokens, if no repair could validate within a reasonable cost threshold, then the candidate with the greatest validation distance was chosen. The repairs were found to be excellent in about 54% of the test cases, good in 33%, and poor in 13%. If we apply the weights that Ripley and Druseikis provide, adjusting the test cases for their likelihood of occurring, then the figures improve to 66% Excellent, 27% Good, 8% Poor. In comparison, the locally least-cost algorithm was found to make a Poor repair in 28% of the cases (using the same costs) [5], the Berkeley Pascal compiler made poor repairs 20% of the time [4], and the much more powerful regionally least-cost algorithm was estimated to make poor repairs 9% of the time[5].

The repair algorithm was implemented with an eye to efficiency, but could not be called "hand-optimized." The program was compiled with the Berkeley Pascal compiler and executed on a Microvax 3600 with Ultrix 3.0. On the Ripley/Druseikis test programs each repair required an average of 0.12 seconds CPU time. By way of comparison, each line of Pascal source required 0.2 seconds to parse. The Ripley/Druseikis programs are perhaps a poor test of speed, since they are so short. We also took a large Pascal program, 5290 lines,

```
function ComputeNonRedundantE(A : V_n; a : V_t; l:level): Rule;
if E(A, a, l − 1) = ? then
    return ?;
repeat
    N := A; lev := l;
    repeat
        if Redundant(E(N, a, lev)) then
            PrevSoln := E(N, a, lev) {Recompute E(N,a,lev) }
        else PrevSoln := E(N, a, lev − 1)
        Cost(E(N, a, lev)) := ∞;
        BeingComputed(N, lev) := true;
        NextLev := −1;
        for each rule r = N → αB^j β in LHS(N) do
            if r = PrevSoln then
                if Redundant(E(B, a, j)) then {force recomputation}
                    NextN := B;
                    NextLev := j;
                    Push(r, lev) onto stack S;
                else if not BeingComputed(B, j) then
                    Replace r with N → αB^{j+1}β;
                    if E(B, a, j + 1) is computed then
                        if Cost(E(N, a, lev)) > Cost(r) then
                            E(N, a, lev) := r;
                    else if not BeingComputed(B, j + 1) then
                        NextN := B;
                        NextLev := j + 1;
                        Push(r, lev) onto stack S;
            else if Cost(E(N, a, lev)) > Cost(r) then
                E(N, a, lev) := r;
        N := NextN;
        lev := NextLev;
    until lev = −1;

    while S ≠ empty do
        Pop (r = D → αB^j β, lev) from S;
        if Cost(E(D, a, lev)) > Cost(r) then
            E(D, a, lev) := r;
until E(A, a, l) = ? or not Redundant(E(A, a, l))
end; {ComputeNonRedundantE}
```

Figure 5: Computation of E-table without redundancy

and inserted 29 syntax errors by hand. On this test, simply parsing a line of source required 0.007 seconds and the average repair took 0.08 seconds. Repairs are much faster in this case because the computed E table entries are retained, and so subsequent repairs may not require addition E table computation. Also, in this test there were no clustered errors, so validation was more successful. On the Ripley/Druseikis programs, the average repair involved 6.47 validation attempts. On the large program, the average repair required only 1.5 validation attempts. Checking for redundant suffixes is currently a bottleneck; this could be improved by a fancier hashing mechanism. Recoding key algorithms in C would also speed the error repair process.

5 Conclusions

Validation works very well with the least-cost repair algorithm. The quality of error repairs is significantly improved; in fact, on the Ripley/Druseikis programs the repair/validation system produces repairs better than any system previously published. The only errors that are not handled satisfactorily are major disturbances such as declarations in the wrong order (**type** before **const**) or comments with improper delimiters. (It is generally acknowledged that these kinds of errors are beyond the capabilities of *any* conventional repair algorithm, though they can be handled by adding "error productions" or by modifying the scanner).

The speed of the repair algorithm is quite good. We can reasonably expect to repair an error in about the amount of time required to scan and parse ten error-free lines. No large E-table need be stored. The incremental computation for a single repair will create the equivalent of just a few rows of the original E-table. The implementer can then trade off time for space by saving the table for possible use in future repairs, or discarding it.

References

[1] C. N. Fischer and R. J. LeBlanc. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, 1988.

[2] C. N. Fischer, D. R. Milton, and J. Mauney. A locally least-cost LL(1) error-corrector. Technical Report Tech. Report 371, University of Wisconsin-Madison, Aug. 1979.

[3] C. N. Fischer, D. R. Milton, and S. B. Quiring. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica*, 13(2):141–154, 1980.

[4] S. L. Graham, C. B. Haley, and W. N. Joy. Practical LR error recovery. *SIGPLAN Notices*, 14(8):168–175, 1979.

[5] J. Mauney. *Least-Cost Error Repair Using Extended Right Context*. PhD thesis, University of Wisconsin-Madison, 1983.

[6] J. Mauney and C. N. Fischer. A forward move algorithm for LL and LR parsers. *SIGPLAN Notices*, 17(6):79–87, June 1982.